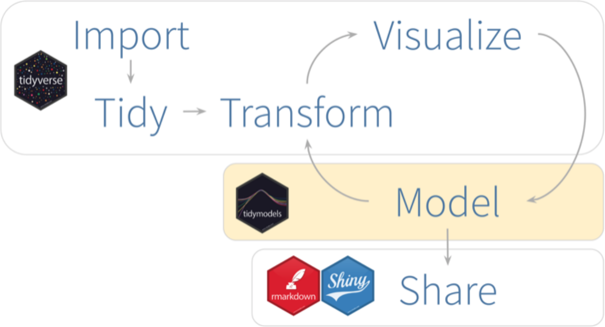
Modeling and Machine Learning in R: **tidymodels**

TIAGO CARVALHO MACHADO DE SOUZA

03/09/2020

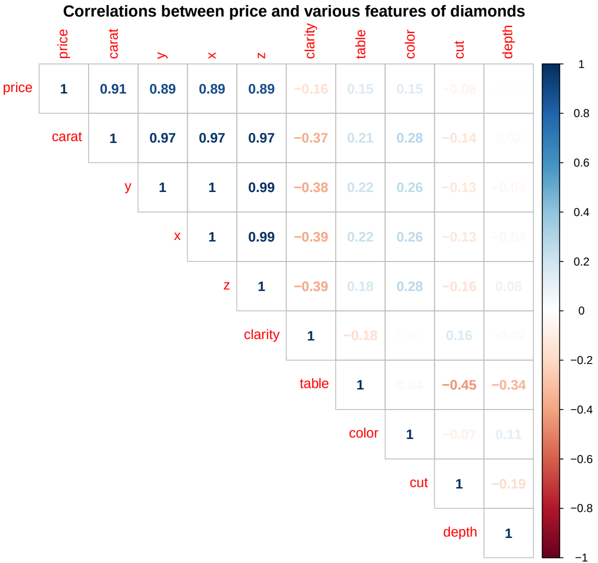
tidymodels within the R Universe



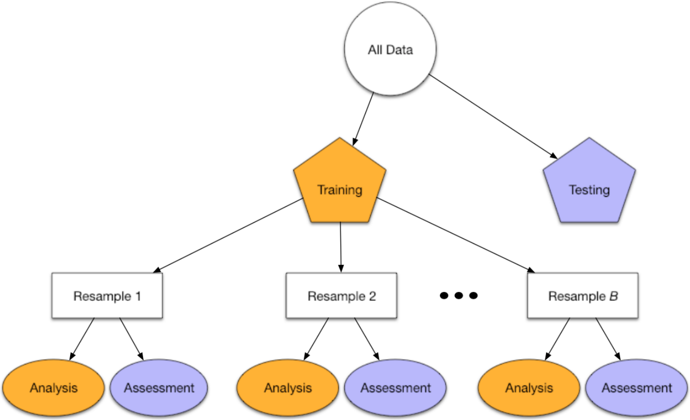
- **tidymodels** is to MODELING what the **tidyverse** is to DATA WRANGLING;
- **tidymodels** has a modular approach: specific, smaller packages are designed to work hand in hand.

tidymodels' main packages

Goal: predict diamond prices



How are we going to do it?



What tools do we have?

Pre-Process → Train → Validate

rsample recipes | parsnip | yardstick

Separating Testing and Training Data

- *rsample* contains a set of functions to create different types of resamples and corresponding classes for their analysis:
  - Traditional resampling techniques for estimating the sampling distribution of a statistic and;
  - Estimating model performance using a holdout set.

Separating Testing and Training Data

```r
dia_split <- initial_split(diamonds, pro = .1, strata = price)

dia_train <- training(dia_split)
dia_test <- testing(dia_split)

dia_vfold <- vfold_cv(dia_train, v = 3, repeats = 1, strata = price)
print(dia_vfold)
```

```
## #  3-fold cross-validation using stratification
## # A tibble: 3 x 2
##   splits              id
##   <list>              <chr>
## 1 <split [3.6K/1.8K]> Fold1
## 2 <split [3.6K/1.8K]> Fold2
## 3 <split [3.6K/1.8K]> Fold3
```

Data Pre-Processing and Feature Engineering

- *recipes* is a method for creating and pre-processing design matrices used for modeling or visualization;
- Idea: define a blueprint that can be used to sequentially define the encodings and pre-processing of the data;
- It is used to prepare a data set (for modeling) using different ‘step_*()’ functions;
- The ‘recipe()’ takes a formula and a data set, and then the different steps are added.

```r
dia_rec <-
    recipe(price ~ ., data = dia_train) %>%
    step_log(all_outcomes()) %>%
    step_normalize(all_predictors(), -all_nominal()) %>%
    step_dummy(all_nominal()) %>%
    step_poly(carat, degree = 2)

prep(dia_rec)
```

```
## Data Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor          9
##
## Training data contained 5395 data points and no missing data.
##
## Operations:
##
## Log transformation on price [trained]
## Centering and scaling for carat, depth, table, x, y, z [trained]
## Dummy variables from cut, color, clarity [trained]
## Orthogonal polynomials on carat [trained]
```

Data Pre-Processing and Feature Engineering

- Calling 'prep()' on a recipe applies all steps;
- Call 'juice()' to extract the transformed data set;
- Call 'bake()' on a new data set.

```
dia_juiced <- juice(prep(dia_rec))
names(dia_juiced)
```

```
##  [1] "depth"        "table"        "x"            "y"            "z"
##  [6] "price"        "cut_1"        "cut_2"        "cut_3"        "cut_4"
## [11] "color_1"      "color_2"      "color_3"      "color_4"      "color_5"
## [16] "color_6"      "clarity_1"    "clarity_2"    "clarity_3"    "clarity_4"
## [21] "clarity_5"    "clarity_6"    "clarity_7"    "carat_poly_1" "carat_poly_2"
```

Defining and Fitting Models

- The goal is to provide a tidy, unified interface to models that can be used to try a range of models without getting bogged down in the syntactical minutiae of the underlying packages;
- Has wrappers around many popular machine learning algorithms, and you can fit then using a unified interface.

Defining and Fitting Models

1. Function specific to each algorithm;
2. ‘set_mode()’ (regression or classification);
3. ‘set_engine()’ back-end/engine/implementation

```
lm_model <-
    linear_reg() %>%
    set_mode("regression") %>%
    set_engine("lm")

print(lm_model)
```

```
## Linear Regression Model Specification (regression)
##
## Computational engine: lm
```

Defining and Fitting Models

- Random Forest: 'ranger' or 'randomForest'?
- How to handle their different interfaces?

```
rand_forest(mtry = 3, trees = 500, min_n = 5) %>%
    set_mode("regression") %>%
    set_engine("ranger", importance = "impurity_corrected")
```

```
## Random Forest Model Specification (regression)
##
## Main Arguments:
##   mtry = 3
##   trees = 500
##   min_n = 5
##
## Engine-Specific Arguments:
##   importance = impurity_corrected
##
## Computational engine: ranger
```

- This example, with a formula. You can also set 'x' and 'y'.

```
lm_fit1 <- fit(lm_model, price ~ ., dia_juiced)
lm_fit1
```

```
## parsnip model object
##
## Fit time:  15ms
##
## Call:
## stats::lm(formula = formula, data = data)
##
## Coefficients:
##  (Intercept)        depth        table            x            y
##    7.731e+00    1.141e-02   -2.867e-03    2.588e-01    5.957e-02
##            z        cut_1        cut_2        cut_3        cut_4
##    4.183e-02    7.431e-02   -7.175e-03   -2.054e-04    3.478e-03
##      color_1      color_2      color_3      color_4      color_5
##   -4.393e-01   -9.030e-02   -1.021e-02    9.078e-03   -1.083e-02
##      color_6    clarity_1    clarity_2    clarity_3    clarity_4
##   -3.402e-03    8.582e-01   -2.314e-01    1.263e-01   -6.051e-02
##    clarity_5    clarity_6    clarity_7  carat_poly_1  carat_poly_2
##    1.986e-02   -6.303e-03    2.120e-02    5.227e+01   -1.782e+01
```

Summarizing Fitted Models

- Takes the messy output of built-in function in R, such as ‘lm’, ‘nls’, and turns them into tidy tibbles;
- From **tidyverse**.

Summarizing Fitted Models

- 'glance()' reports information about the entire model;
- 'tidy()' summarizes information about model components.

```
glance(lm_fit1$fit)
```

```
## # A tibble: 1 x 11
##   r.squared adj.r.squared sigma statistic p.value    df logLik    AIC    BIC
##       <dbl>         <dbl> <dbl>     <dbl>   <dbl> <int>  <dbl>  <dbl>  <dbl>
## 1     0.979         0.979 0.150    10274.       0    25  2598. -5143. -4972.
## # ... with 2 more variables: deviance <dbl>, df.residual <int>
```

```
tidy(lm_fit1) %>%
    arrange(desc(abs(statistic))) %>%
    print()
```

```
## # A tibble: 25 x 5
##    term          estimate std.error statistic  p.value
##    <chr>            <dbl>     <dbl>     <dbl>    <dbl>
##  1 (Intercept)    7.73      0.00421    1838.   0.
##  2 carat_poly_2 -17.8       0.257       -69.2  0.
##  3 clarity_1      0.858     0.0130       66.0  0.
##  4 color_1       -0.439     0.00718     -61.2  0.
##  5 carat_poly_1  52.3       1.10         47.3  0.
##  6 clarity_2     -0.231     0.0122      -19.0  3.71e-78
##  7 color_2       -0.0903    0.00653     -13.8  1.04e-42
##  8 clarity_3      0.126     0.0104       12.2  1.32e-33
##  9 cut_1          0.0743    0.00970       7.66 2.15e-14
## 10 clarity_4     -0.0605    0.00820      -7.38 1.82e-13
## # ... with 15 more rows
```

Summarizing Fitted Models

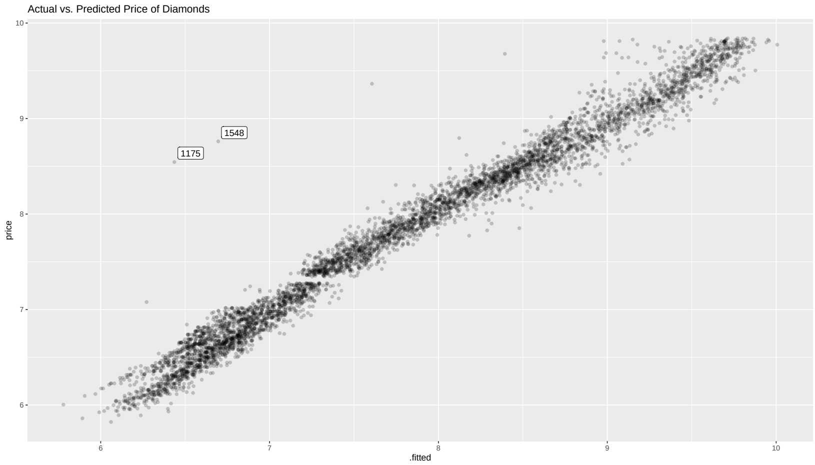
- 'augment()' is used to get model predictions, residuals, etc.

```
lm_predicted <- augment(lm_fit1$fit, data = dia_juiced) %>%
    rowid_to_column()
print(select(lm_predicted, rowid, price, .fitted:.std.resid))
```

```
## # A tibble: 5,395 x 9
##    rowid price .fitted .se.fit  .resid    .hat .sigma   .cooksd .std.resid
##    <int> <dbl>   <dbl>   <dbl>   <dbl>   <dbl>  <dbl>     <dbl>      <dbl>
##  1     1  5.82    6.06 0.0106  -0.238  0.00501  0.150 0.000512      -1.59
##  2     2  5.86    5.89 0.0136  -0.0312 0.00825  0.150 0.0000145     -0.209
##  3     3  6.00    6.08 0.00949 -0.0772 0.00401  0.150 0.0000429     -0.516
##  4     4  6.00    6.21 0.0103  -0.204  0.00472  0.150 0.000352      -1.36
##  5     5  6.00    5.78 0.0126   0.225  0.00702  0.150 0.000639       1.50
##  6     6  6.32    6.55 0.0112  -0.239  0.00561  0.150 0.000577      -1.60
##  7     7  6.32    6.17 0.00900  0.142  0.00361  0.150 0.000131       0.953
##  8     8  6.32    6.18 0.00869  0.135  0.00336  0.150 0.000111       0.905
##  9     9  6.32    6.55 0.00980 -0.229  0.00428  0.150 0.000402      -1.53
## 10    10  7.92    7.75 0.00778  0.176  0.00270  0.150 0.000150       1.18
## # ... with 5,385 more rows
```

Visualizing Results

```
ggplot(lm_predicted, aes(.fitted, price)) +
    geom_point(alpha = .2) +
    ggrepel::geom_label_repel(aes(label = rowid),
        data = filter(lm_predicted, abs(.resid) > 2)) +
    labs(title = "Actual vs. Predicted Price of Diamonds")
```



Evaluating Model Performance

Evaluating Model Performance

- Use 'rsample', 'parsinp' and 'yardstick' for cross-validation (3).

```
print(dia_vfold)
```

```
## #  3-fold cross-validation using stratification
## # A tibble: 3 x 2
##   splits               id
##   <list>               <chr>
## 1 <split [3.6K/1.8K]> Fold1
## 2 <split [3.6K/1.8K]> Fold2
## 3 <split [3.6K/1.8K]> Fold3
```

- Extract analysis/training and assessment/testing data.

```
lm_fit2 <- mutate(dia_vfold,
                  df_ana = map(splits, analysis),
                  df_ass = map(splits, assessment))
print(lm_fit2)
```

```
## #  3-fold cross-validation using stratification
## # A tibble: 3 x 4
##   splits               id    df_ana                df_ass
##   <list>               <chr> <list>                <list>
## 1 <split [3.6K/1.8K]> Fold1 <tibble [3,596 x 10]> <tibble [1,799 x 10]>
## 2 <split [3.6K/1.8K]> Fold2 <tibble [3,596 x 10]> <tibble [1,799 x 10]>
## 3 <split [3.6K/1.8K]> Fold3 <tibble [3,598 x 10]> <tibble [1,797 x 10]>
```

Evaluating Model Performance

- Prepare data / fit model / predict.

```
lm_fit2 <-
    lm_fit2 %>%
    # prep, juice, bake
    mutate(
        recipe = map(df_ana, ~prep(dia_rec, training = .x)),
        df_ana = map(recipe, juice),
        df_ass = map2(recipe, df_ass, ~bake(.x, new_data = .y))
    ) %>%
    #fit
    mutate(
        model_fit = map(df_ana, ~fit(lm_model, price ~ ., data = .x))
    ) %>%
    # predict
    mutate(
        model_pred = map2(model_fit, df_ass, ~predict(.x, new_data = .y))
    )

print(select(lm_fit2, id, recipe:model_pred))
```

```
## # A tibble: 3 x 4
##   id    recipe   model_fit model_pred
##   <chr> <list>   <list>    <list>
## 1 Fold1 <recipe> <fit[+]>  <tibble [1,799 x 1]>
## 2 Fold2 <recipe> <fit[+]>  <tibble [1,799 x 1]>
## 3 Fold3 <recipe> <fit[+]>  <tibble [1,797 x 1]>
```

Evaluating Model Performance

- Select original and predicted values.

```
lm_preds <-
    lm_fit2 %>%
    mutate(res = map2(df_ass, model_pred, ~data.frame(price = .x$price,
                                                      .pred = .y$.pred))
    ) %>%
    select(id, res) %>%
    tidyr::unnest(res) %>%
    group_by(id)

print(lm_preds)
```

```
## # A tibble: 5,395 x 3
## # Groups:   id [3]
##    id    price .pred
##    <chr> <dbl> <dbl>
##  1 Fold1  6.00  6.09
##  2 Fold1  6.00  6.23
##  3 Fold1  6.00  5.79
##  4 Fold1  6.32  6.19
##  5 Fold1  7.93  7.82
##  6 Fold1  7.93  7.86
##  7 Fold1  7.93  7.76
##  8 Fold1  7.94  8.00
##  9 Fold1  7.94  7.78
## 10 Fold1  7.94  7.79
## # ... with 5,385 more rows
```

Evaluating Model Performance

- 'metrics()' has default measures for numeric and categorical outcomes (numeric - 'rmse', 'rsq', 'mae');
- You can choose other if you'd like with 'metric_set()'.

```
print(metrics(lm_preds, truth = price, estimate = .pred))
```

```
## # A tibble: 9 x 4
##   id    .metric .estimator .estimate
##   <chr> <chr>   <chr>          <dbl>
## 1 Fold1 rmse    standard       0.151
## 2 Fold2 rmse    standard       0.148
## 3 Fold3 rmse    standard       0.314
## 4 Fold1 rsq     standard       0.979
## 5 Fold2 rsq     standard       0.979
## 6 Fold3 rsq     standard       0.911
## 7 Fold1 mae     standard       0.116
## 8 Fold2 mae     standard       0.114
## 9 Fold3 mae     standard       0.111
```

Tuning Model Parameters

- 'tune' wants to facilitate hyper-parameter tuning for the **tidymodels** packages;
- 'dials' contains tools to create and manage values of tuning parameters;
- Let's tune the 'mtry' and 'degree' parameters.

Tuning Model Parameters

- Preparing a 'parsnip' Model for tuning.

```
rf_model <-
    rand_forest(mtry = tune()) %>%
    set_mode("regression") %>%
    set_engine("ranger")

print(parameters(rf_model))
```

```
## Collection of 1 parameters for tuning
##
##    id parameter type object class
##  mtry           mtry    nparam[?]
##
## Model parameters needing finalization:
##    # Randomly Selected Predictors ('mtry')
##
## See '?dials::finalize' or '?dials::update.parameters' for more information.
```

Tuning Model Parameters

- Preparing Data for Tuning: ‘recipes’;
- Tune the degree of the polynomial for the variable ‘carat’.

```r
dia_rec2 <-
    recipe(price ~ ., data = dia_train) %>%
    step_log(all_outcomes()) %>%
    step_normalize(all_predictors(), -all_nominal()) %>%
    step_dummy(all_nominal()) %>%
    step_poly(carat, degree = tune())

dia_rec2 %>%
    parameters() %>%
    pull("object") %>%
    print()
```

```
## [[1]]
## Polynomial Degree  (quantitative)
## Range: [1, 3]
```

Combine Everything

- Object that can bundle together pre-processing, modeling and post-processing requests;
- The recipe prepping and model fitting can be executed using a single call to 'fit()'.

Combine Everything

```
rf_wflow <-
    workflow() %>%
    add_model(rf_model) %>%
    add_recipe(dia_rec2)

print(rf_wflow)
```

```
## == Workflow ==================================================================
## Preprocessor: Recipe
## Model: rand_forest()
##
## -- Preprocessor --------------------------------------------------------------
## 4 Recipe Steps
##
## * step_log()
## * step_normalize()
## * step_dummy()
## * step_poly()
##
## -- Model ---------------------------------------------------------------------
## Random Forest Model Specification (regression)
##
## Main Arguments:
##   mtry = tune()
##
## Computational engine: ranger
```

Tuning Parameters

- Update the parameters in the workflow;
- Cross-validation for tuning: select the best combination of hyper-parameters.

```
rf_param <-
    rf_wflow %>%
    parameters() %>%
    update(mtry = mtry(range = c(3L, 5L)),
           degree = degree_int(range = c(2L, 4L)))

print(rf_param$object)
```

```
## [[1]]
## # Randomly Selected Predictors  (quantitative)
## Range: [3, 5]
##
## [[2]]
## Polynomial Degree  (quantitative)
## Range: [2, 4]
```

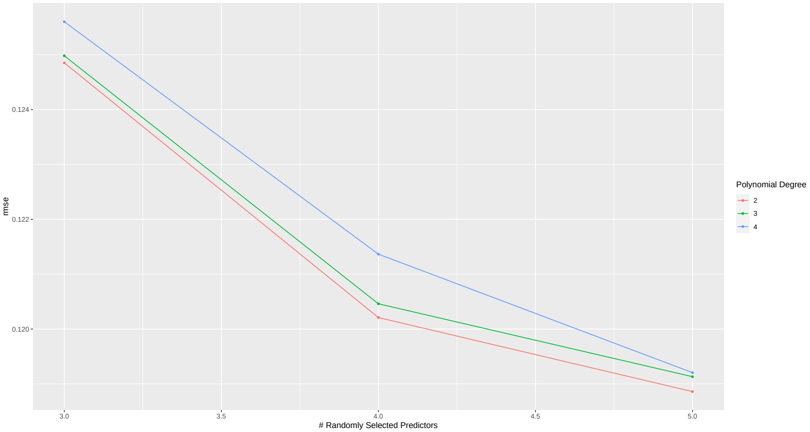
Tuning Parameters

```r
rf_grid <- grid_regular(rf_param)

print(rf_grid)
```

```
## # A tibble: 9 x 2
##    mtry degree
##   <int>  <int>
## 1     3      2
## 2     4      2
## 3     5      2
## 4     3      3
## 5     4      3
## 6     5      3
## 7     3      4
## 8     4      4
## 9     5      4
```

Tuning Parameters

```r
rf_search <- tune_grid(rf_wflow,
                       grid = rf_grid,
                       resamples = dia_vfold,
                       param_info = rf_param)

autoplot(rf_search, metric = "rmse") +
    labs("Results of Grid Search for Two Tuning Parameters of a Random Forest")
```



Model Selection

```
print(show_best(rf_search, "rmse", 9))
```

```
## # A tibble: 9 x 8
##    mtry degree .config         .metric .estimator  mean     n std_err
##   <int>  <int> <chr>           <chr>   <chr>      <dbl> <int>   <dbl>
## 1     5      2 Recipe1_Model3 rmse    standard   0.119     3 0.00377
## 2     5      3 Recipe2_Model3 rmse    standard   0.119     3 0.00349
## 3     5      4 Recipe3_Model3 rmse    standard   0.119     3 0.00362
## 4     4      2 Recipe1_Model2 rmse    standard   0.120     3 0.00385
## 5     4      3 Recipe2_Model2 rmse    standard   0.120     3 0.00390
## 6     4      4 Recipe3_Model2 rmse    standard   0.121     3 0.00375
## 7     3      2 Recipe1_Model1 rmse    standard   0.125     3 0.00373
## 8     3      3 Recipe2_Model1 rmse    standard   0.125     3 0.00362
## 9     3      4 Recipe3_Model1 rmse    standard   0.126     3 0.00390
```

```
print(select_best(rf_search, metric = "rmse"))
```

```
## # A tibble: 1 x 3
##    mtry degree .config
##   <int>  <int> <chr>
## 1     5      2 Recipe1_Model3
```

```
print(select_by_one_std_err(rf_search, mtry, degree, metric = "rmse"))
```

```
## # A tibble: 1 x 10
##    mtry degree .config        .metric .estimator  mean     n std_err .best .bound
##   <int>  <int> <chr>          <chr>   <chr>      <dbl> <int>   <dbl> <dbl>  <dbl>
## 1     4      2 Recipe1_Mode~ rmse    standard   0.120     3 0.00385 0.119  0.123
```

Best Model and Final Predictions

```
rf_param_final <- select_by_one_std_err(rf_search, mtry, degree, metric = "rmse")

rf_wflow_final <- finalize_workflow(rf_wflow, rf_param_final)

rf_wflow_final_fit <- fit(rf_wflow_final, data = dia_train)
```

- Want to use ‘predict()’ on data never seem before (‘dia_test’);
- However, it does not work, because the outcome is modified in the recipe via ‘step_log()’.

Best Model and Final Predictions

- Workaround:
  1. Prepped recipe is extracted from the workflow;
  2. This is used to 'bake()' the testing data;
  3. Use this baked data set together with extracted model for final predictions.

```
dia_rec3 <- pull_workflow_prepped_recipe(rf_wflow_final_fit)
rf_final_fit <- pull_workflow_fit(rf_wflow_final_fit)

dia_test$.pred <- predict(rf_final_fit,
                          new_data = bake(dia_rec3, dia_test))$.pred
dia_test$logprice <- log(dia_test$price)

metrics(dia_test, truth = logprice, estimate = .pred)
```

```
## # A tibble: 3 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard      0.114
## 2 rsq     standard      0.987
## 3 mae     standard      0.0848
```